Natural Language Understanding and Communication for Multi-Agent Systems

Sean Trott, Aurélien Appriou, Jerome Feldman, Adam Janin
International Computer Science Institute
1947 Center Street, #600
Berkeley, CA 94704
seantrott@icsi.berkeley.edu appriou.aurelien@berkeley.edu feldman@icsi.berkeley.edu janin@icsi.berkeleky.edu

Abstract

Natural Language Understanding (NLU) studies machine language comprehension and action without human intervention. We describe an implemented system that supports deep semantic NLU for controlling systems with multiple simulated robot agents. The system supports bidirectional communication for both human-agent and agent-agent interaction. This interaction is achieved with the use of N-tuples, a novel form of Agent Communication Language using shared protocols with content expressing actions or intentions. The system's portability and flexibility is facilitated by its division into unchanging "core" and "application-specific" components.

Introduction

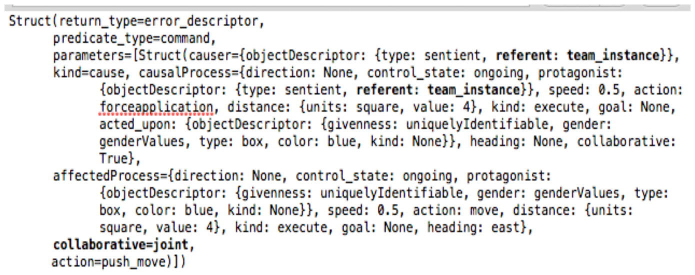
Historically, controlling multi-agent systems has been difficult, particularly for systems involving both human agents and autonomous or semi-autonomous robotic agents. Multi-agent problems introduce new difficulties, including the sharing of world information and solving problems by planning or coordinating collaborative actions (Taylor et al. 2011).

Much previous research in the field has focused on direct planning without agent collaboration (Shi et al. 2014), while other work has incorporated more collaborative elements (Allen and Perrault 1980), (Ferguson and Allen 2011), (Subramanian, Kumar, and Cohen 2006).

We have implemented an approach based on our previous work on Natural Language Understanding (Fig. 2) (Khayrallah, Trott, and Feldman 2015). This is grounded in cognitive linguistics research on deep embodied semantics with a focus on the semantics of action (Feldman 2007), (Feldman, Dodge, and Bryant 2009) and uses Embodied Construction Grammar (ECG) and the ECG Analyzer (Bryant 2008) as the front-end for the natural language interface. All grammar development has been greatly aided by the ECG Workbench, an Eclipse-based application used for grammar design and testing. More information is available at:
http://www.icsi.berkeley.edu/icsi/projects/ai/ntl.

We have filed a patent application on this development, LCAS (Language Communication with Autonomous Systems). This paper extends the previous work with demonstration and discussion of the system architecture, including for multi-agent communication.

In the LCAS system, human-agent and agent-agent interactions are facilitated by N-tuples (Fig. 1), which contain action protocols and semantic content. Our goal is to develop a framework whereby all agents involved in an application can send and receive information in the same form. The mechanism of universal, but customizable, agent IPC is grounded in general *N-tuple Templates*. The template underlying Figure 1 consists of the argument names followed by type restrictions. Mapping from natural language input to completed N-tuples like Figure 1 is the key NLU step in the system, as will be discussed below.



```
Struct(return_type=error_descriptor,
      predicate_type=command,
      parameters=[Struct(causer={objectDescriptor: {type: sentient, referent: team_instance}},
      kind=cause, causalProcess={direction: None, control_state: ongoing, protagonist:
            {objectDescriptor: {type: sentient, referent: team_instance}}, speed: 0.5, action:
            forceapplication, distance: {units: square, value: 4}, kind: execute, goal: None,
            acted_upon: {objectDescriptor: {givenness: uniquelyIdentifiable, gender:
            genderValues, type: box, color: blue, kind: None}}, heading: None, collaborative:
            True},
      affectedProcess={direction: None, control_state: ongoing, protagonist:
            {objectDescriptor: {givenness: uniquelyIdentifiable, gender: genderValues, type:
            box, color: blue, kind: None}}, speed: 0.5, action: move, distance: {units:
            square, value: 4}, kind: execute, goal: None, heading: east},
      collaborative=joint,
      action=push_move)])
```

Figure 1: N-tuple "Team, push the blue box east together!"

System Architecture

The major components are shown in Fig. 2, which depicts the layout of the components for a simulated robot application with a single Agent/Problem Solver. A major design goal is modularity, which we implement at two levels. At the component level, the subsystems run as independent programs communicating through well-defined protocols. Although some code, such as the network communication library, is shared among the components, they run independently and potentially on separate machines. More details on the individual components are described later in the paper.

Another important level of modularity comes from separating the components into **Core** and **App** parts. For each component, the **Core** contains the parts of the component that do not change from application to application. To retarget the system to a new App, several steps of system integration are needed, as will be discussed below. The **App** parts comprise components that must be modified by the system integrator when the application changes. We believe that this separation will allow optimal repurposing – some work must be done to integrate a new application, but the majority of the language side need only be written once in Core. The tractability of writing the language Core only once arises from the restricted domain of control of autonomous systems. General NLU across domains and tasks remains intractable.

The loose coupling between the language front-end and the action (Fig. 2) allows the system to be retargeted for a variety of tasks. Here, our implementation connects language to a simulated robot, but ECG has also been used as the language model for other tasks (Oliva et al. 2012).

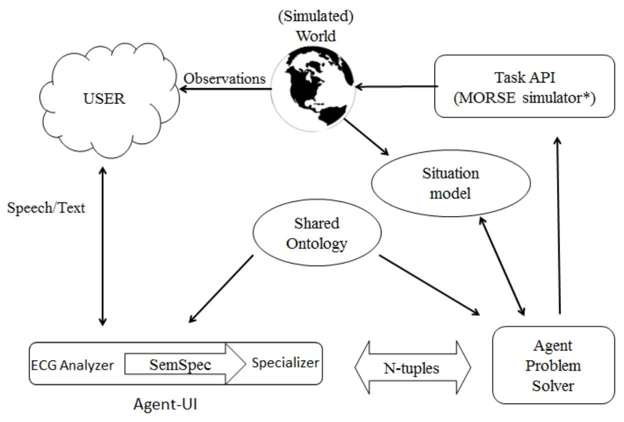


Figure 2: System architecture. This API references the MORSE simulator but other applications could be substituted.

Communication Paradigm: N-tuples

A key feature of our system is the shared ability across all agents to send and receive N-tuples. Language input from a human user is converted into N-tuples, and agents communicate with each other (to share information and plan collaborative actions) and the human (when asking for clarification, etc.) through the use of N-tuples. All avenues of N-tuple communication are bidirectional.

Agent communication to the human operator is done using N-tuple information in pattern-based natural language generation (e.g., “which box?” or “I have discovered a new box at location *x*, *y*”). For most interaction purposes, we feel the pattern-based responses are sufficient. In fact, there are informal studies that suggest that habitability is improved by making the system responses less human-like.

A crucial part of our design is that agent-agent communication can be accomplished solely through N-tuples, without the need to generate natural language or another type of agent communication language. The design does not suggest that the N-tuples used for Agent-Agent communication should be translated into natural language. One should not expect end users to monitor the detailed interaction of agents. Any language interaction with a user is channeled through the Agent-UI as shown. Of course, an App will likely interact with a user non-linguistically, such as by modifying displays or other HCI capabilities. For example, in the pilot system, the Morse App (using Blender) provides realistic action displays.

N-tuple Implementation

The communication of N-tuples is provided by a Core facility known as Transport. Typically, each agent (the Agent-UI and each Agent/Problem Solver) instantiates one Transport using the agent's name (e.g. the Agent-UI Transport is called "Agent-UI"; the top-level Problem Solver is called "Boss-Agent"). An agent can then send an N-tuple to any other agent (Fig. 4). It is also possible for multiple agents to share a transport with the same name. For example, Agent 1 and Agent 2 could both receive N-tuples sent to the Transport "Team-Agent".

The current implementation of Transport requires no setup configuration – all agents automatically and transparently announce themselves on the local area network. In practice, it is likely that sites will require a component that implements the Transport API following their own security and authentication policies.

Language Front-End: Analyzer/Specializer

A human user gives a natural language command to an Agent-UI (User-Interface Agent). The Agent-UI is responsible for mediating communication between the human agent and the agents involved in the application.

Specifically, the Agent-UI (Figure 2) receives speech or text input from a human user, performs a deep semantic analysis using the ECG Analyzer (Bryant 2008) which outputs and produces a SemSpec as shown in the lower left of Figure 2. This operation is described in more detail in the 2015 paper (Khayrallah, Trott, and Feldman 2015). The SemSpec is intended to be independent of any application, but we have not yet tested this extensively.

The App-dependent aspects of the language analysis are handled by the Specializer, which crawls the SemSpec structure and outputs an N-tuple (like Fig. 1) for the Problem Solver. The Agent-UI then sends the N-tuple to the Problem Solver. Task relevant information is defined declaratively in the N-tuple templates, which are shared between the Specializer and the Problem Solver.

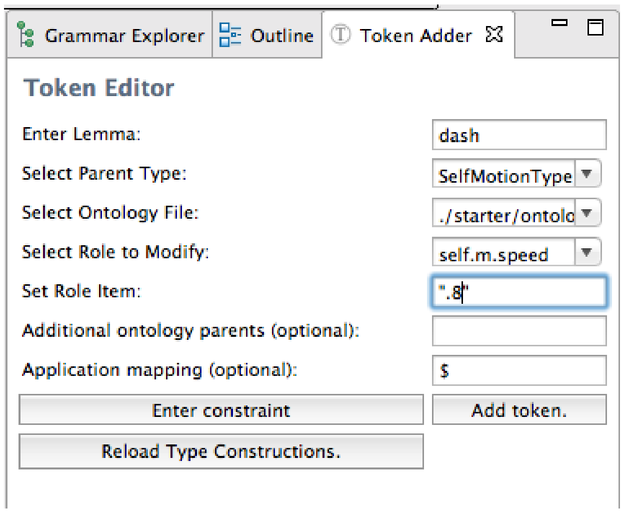
The Agent-UI is capable of receiving feedback from the Problem Solver (as a different N-tuple), as well as interacting with the user through speech or text. As mentioned above, natural language generation is accomplished with the use of pattern-based responses, which are geared towards the types of information or dialogs a human might need to hear. The most common forms of interaction are requests for clarification (if the user input is underspecified), responses to queries, or notifications of object discovery.

Front-End Modularity

Both the Analyzer and the Specializer are divided into Core and App sides. The Analyzer core contains grammars, schemas, ontologies, and code that support the subset of a language related to control of autonomous systems; most of our work has been in English, but we have also implemented the robot demo in Spanish and French.

A system integrator adds domain-specific words and ontology entries relevant to the application to the Analyzer App. For example, the Analyzer Core has grammar and words for commanding an autonomous system to move. The system integrator could add a specific term "dash" meaning to move at the robot's top speed. The speed of “dash” is application-specific, so must be implemented by the App side. Adding words and ontology entries will be quite common, and we provide a specific user interface called the Token Editor (Fig. 3) to make it easy. If the application requires grammar that isn't in Core, the application integrator must add it to Core, using the ECG Workbench. We believe that this will be rare.

The Specializer Core contains code that traverses the SemSpec produced by the Analyzer component. The Specializer-App includes the Core, as well as task-relevant N-tuple templates and code that specify how to build N-tuples from a SemSpec. As mentioned above, these templates are shared between the Specializer and Problem Solver; this establishes a shared set of vocabulary and communication language between the two modules. The values in an N-tuple (Fig. 1) are all ontology values. A key part of our design is a “shared ontology” (Fig. 2) between the language and application sides; the Token Editor (Fig. 3) can be used to specify mappings from the text to language and application ontologies.



Token Editor

Enter Lemma:	dash
Select Parent Type:	SelfMotionType
Select Ontology File:	./starter/ontolo
Select Role to Modify:	self.m.speed
Set Role Item:	".8"
Additional ontology parents (optional):	
Application mapping (optional):	$

Figure 3: Token Editor, used to add "dash" and other tokens to grammar.

Action: Problem Solver

The Problem Solver uses information from an N-tuple, as well as its world model, to make decisions and solve complex problems. It makes API calls to the underlying application – in this case, the MORSE robot simulator (Echeverria et al. 2011). A Problem Solver’s capabilities are inherently dependent upon the constraints of the underlying application. As such, the Problem Solver core is quite small, consisting mainly of code to scan, build, and transmit N-tuples, as well as answering queries and requesting clarification from the human by communicating with the Agent-UI, as discussed in the earlier 2015 paper (Khayrallah, Trott, and Feldman 2015).

In this case, the MORSE simulator offers realistic motion physics, as well as various proximity sensors, which are used in detecting or discovering new objects. If new information is gleaned about the world (by contact in the current demo), the Problem Solver communicates this information in an N-tuple to the Agent-UI, which informs the user via speech or text.

Part of our new multi-agent system is the development of multiple levels of complexity, which involve more than one Problem Solver.

Levels of Multi-Agent Complexity

The first level of complexity uses only one Problem Solver and involves only one Agent Problem Solver (as depicted in Fig. 2), which has one world model. If a robot discovers a new object in the world, the shared world model is updated. The Agent Problem Solver communicates all requests and other N-tuples back to the Agent-UI and instructions to application modules.

The second level of complexity uses multiple Problem Solvers (Fig. 4). There is a Boss-Agent, and there are also Application-Agents, each with its own separate world model. The Boss-Agent contains its own world model. The Boss-Agent receives incoming N-tuples from the Agent-UI and conveys them to the Application-Agents. The Boss-Agent also performs high-level problem solving, such as determining which Application-Agent is best suited for a given task. In this model, there is no communication between the individual Application-Agents, but Application-Agents and Boss-Agents communicate with each other using N-tuples.

The third level of complexity, like the second, involves a Boss-Agent and individual Application-Agents (Fig. 4). The chief difference is that Application-Agents can also send N-tuples to each other and coordinate actions among themselves, as depicted by the arrow in the lower right of the diagram below (Fig. 4).

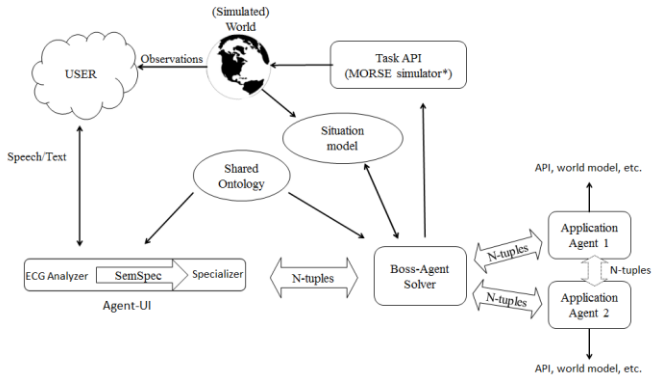


Figure 4: System architecture for second and third level of complexity.

Applications

Our current demonstration involves two simulated robots ("Robot1" and "Robot2"), and boxes of varying color and size. The user can address a particular robot, such as "Robot1, dash behind the blue box!" In this case, the Boss-Agent routes the resulting N-tuple to one of the Application-Agents.

The user can also address the system as a whole using the word "Team", such as: "Team, push the blue box east together!" The Analyzer produces a SemSpec as usual, and the Specializer's resulting N-tuple (Fig. 1) notes that the "protagonist" of the process is "*team_instance*". The N-tuple also notes that the process is meant to be collaborative.

In this case, the Boss-Agent must perform high-level problem solving. This process is also collaborative, meaning the Application-Agents should work together to solve it. The Boss-Agent composes its own N-tuples from the command, which contain detailed instructions, and dispatches versions to each Application-Agent. Ultimately, the two robots move into position and use their combined power to push the blue box east.

The Boss-Agent must also make high-level decisions if the user's instructions are less precise, such as: "Team, push the blue box east!" In this case, the user is requesting the same desired effect – the blue box being pushed east – but specifies neither the actual agent ("Robot1" or "Robot2") or whether the process should be done collaboratively. The Boss-Agent takes into account various factors from its world model to decide how the task ought to be executed. For example, if Robot1 is closer to the blue box, the Boss-Agent might build and dispatch an N-tuple to Robot1 ordering it to push the box east. However, if Robot1 is occupied with another task, or is unable to perform this task, the Boss-Agent would select Robot2 and send a customized N-tuple with instructions for action (Fig. 5). Alternatively, if the box was too heavy for one robot to push, the Boss-Agent would order the robots to work collaboratively. Of course, the factors taken into consideration by the Boss-Agent are contingent on the task and application domain.

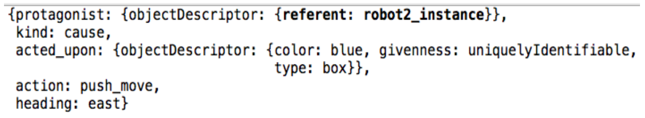

```
{protagonist: {objectDescriptor: {referent: robot2_instance}},
 kind: cause,
 acted_upon: {objectDescriptor: {color: blue, givenness: uniquelyIdentifiable,
                                 type: box}},
 action: push_move,
 heading: east}
```

Figure 5: Sample N-tuple fragment sent from Boss-Agent to Application-Agent (Robot2).

A video demonstration of this example can be found at: https://www.youtube.com/watch?v=46jYgBIw_VA.

A compilation video demonstrating various features of our robot system can be found at:
https://www.youtube.com/watch?v=mffl4-FqZaU.

Limitations and Future Work

Currently, communication from the Boss-Agent back to the user – such as requests for clarification – is translated from an N-tuple into a "pattern" response, such as: "which *red box*?" The addition of more complex natural language generation might improve the system.

In an attempt to modularize the system further, we are implementing a model that communicates N-tuples across different network channels; this would be invaluable for situations such as communicating with an autonomous submarine from land.

Finally, we are integrating a speech front-end to the language module.

Acknowledgments

This work is supported by the Office of Naval Research grant number N000141110416.

References

Taylor, M. E., Jain, M., Kiekintveld, C., Kwak, J. Y., Yang, R., Yin, Z., & Tambe, M. (2011). Two decades of multiagent teamwork research: past, present, and future. In Collaborative Agents-Research and Development (pp. 137-151). Springer Berlin Heidelberg.

Shi, Z., Zhang, J., Yue, J., & Yang, X. (2014). A Cognitive Model for Multi-Agent Collaboration. International Journal of Intelligence Science, 4(01), 1-6.

Allen, J. F., & Perrault, C. R. (1980). Analyzing intention in utterances. Artificial Intelligence, 15(3), 143-178.

Ferguson, G., & Allen, J. F. (2011). A Cognitive Model for Collaborative Agents. In AAAI Fall Symposium: Advances in Cognitive Systems.

Subramanian, R. A., Kumar, S., & Cohen, P. R. (2006). Integrating joint intention theory, belief reasoning, and communicative action for generating team-oriented dialogue. In Proceedings of the National Conference on Artificial Intelligence (Vol. 21, No. 2, p. 1501).

Khayrallah, H., Trott, S., & Feldman, J. (2015). Natural Language For Human Robot Interaction. Proceedings of the Workshop on Human-Robot Teaming at the 10th ACM/IEEE International Conference on Human-Robot Interaction, Portland, Oregon.

Feldman, J., Dodge, E., & Bryant, J. (2009). A neural theory of language and embodied construction grammar. The Oxford Handbook of Linguistic Analysis., 111 —- 138.

Feldman, J. (2007). From Molecule to Metaphor. A Neural theory of Language. *The MIT Press*.

Bryant, J. E. (2008). Best-Fit Construction Analysis. Analysis.

Oliva, J., Feldman J., Giraldi L., and Dodge E. (2012) Ontology Driven Contextual Reference Resolution in Embodied Construction Grammar. In the proceedings of the 7th Annual Constraint Solving and Language Processing Workshop. Orléans, France.

Echeverria, G.; Lassabe, N.; Degroote, A. and Lemaignan, S. (2011). Modular open robots simulation engine: Morse. In the proceedings of the 2011 IEEE International Conference Robotics and Automation, 46-51 IEEE.